**Pintos Project 1: User Program (1)**

담당 교수 : 박성용 교수님

조 / 조원 : 20181662 이건영

개발 기간 : 2022.09.18~2022.10.03

1. **개발 목표**

Pintos 내에서 stack과 system call handler 등을 구현하여 system call이 동작하게 하고, 사용자 정의 system call을 구현하여 본다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

User가 입력한 arguments를 공백을 기준으로 parsing 하여 80x86 Calling Convention에 맞추어 stack에 저장한다.

1. User Memory Access

User program이 invalid한 pointer에 access한 경우, page fault를 방지하기 위해 미리 검사하여 안전하게 exit 하도록 한다.

1. System Calls

System call handler를 구현하여 전달받은 system call number에 해당하는 기능을 수행하도록 구현한다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술 (기타 내용은 서술하지 않아도 됨.)**
* Argument Passing
  + 커널 내 스택에 argument를 쌓는 과정 설명
* User Memory Access
  + Pintos 상에서의 invalid memory access 개념을 간략히 설명

User program은 자신에게 mapping된 virtual memory space에만 접근하여야 한다. 그러나 만약 NULL pointer를 사용하여 메모리에 접근하거나, 자신에게 mapping 되지 않은 virtual memory에 접근하거나, kernel address space에 접근한다면 invalid memory access가 된다.

* + Invalid memory access를 어떻게 막을 것인지 설명

이를 막기 위해서는 현재 접근하고자 하는 메모리 영역이 자신에게 mapping 되었는지 확인하는 절차가 필요하다. 따라서 이를 vaddr.h 내의 함수를 이용해서 검사해주면 된다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

User가 OS와 kernel이 제공하는 기능을 API화 하여 편하고 안전하게 사용할 수 있도록 도와 주기 위함이다.

* + 이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명 (하나의 시스템 콜 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)

halt : shutdown\_power\_off 함수를 호출한다.

exit : status를 인자로 받아 종료된 thread의 이름과 상태를 출력하고, thread의 상태를 업데이트 해준 후, thread\_exit 함수를 호출한다.

exec : file 명을 전달받아 process\_execute 함수를 실행한다. thread\_create 함수로 thread를 생성하여 start\_process 함수를 실행한다. 그 return 값인 pid를 return한다.

wait : pid를 전달받아 process\_wait 함수를 호출한다. semaphore를 이용해 child process의 실행과 종료, child list에서의 제거를 기다려준 후 exit\_status를 return 한다.

read : stdin에서 들어오는 내용을 input\_getc 함수를 통해 받아온다.

write : stdout에 putbuf 함수를 이용해 버퍼의 내용을 출력한다.

Fibonacci : 입력받은 n번째 피보나치 수열의 값을 출력한다.

max\_of\_four\_int : 입력받은 네 수의 최댓값을 출력한다.

* + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명

User가 system call을 호출하면 해당 system call이 syscallN(N은 argument의 개수)를 호출한다. 이 syscallN은 system call number와 argument를 stack에 push하고, int $0x30 instruction을 통해 interrupt를 발생시킨다. interrupt handler는 syscall handler를 호출하고, syscall handler는 stack의 정보들을 이용하여 kernel이 알맞은 기능을 수행하도록 한다. 이 return 값은 eax 레지스터에 저장되어 User가 받아서 사용할 수 있다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* **II. A.의 개발 범위를 포함하여 구현 내용에 대한 일정 작성**

22.09.18~22.09.24 : Argument parsing 및 stack construction 구현

22.09.24~22.09.28 : System call handler 구현

22.09.29~22.09.30 : Additional system call 구현

* 1. **개발 방법**
* **II. B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

Argument parsing : userprog/process.c 의 load 함수 부분에 file\_name을 parsing 하는 부분을 추가하고, setup\_stack 이후 stack에 내용물을 push하는 부분을 추가한다.

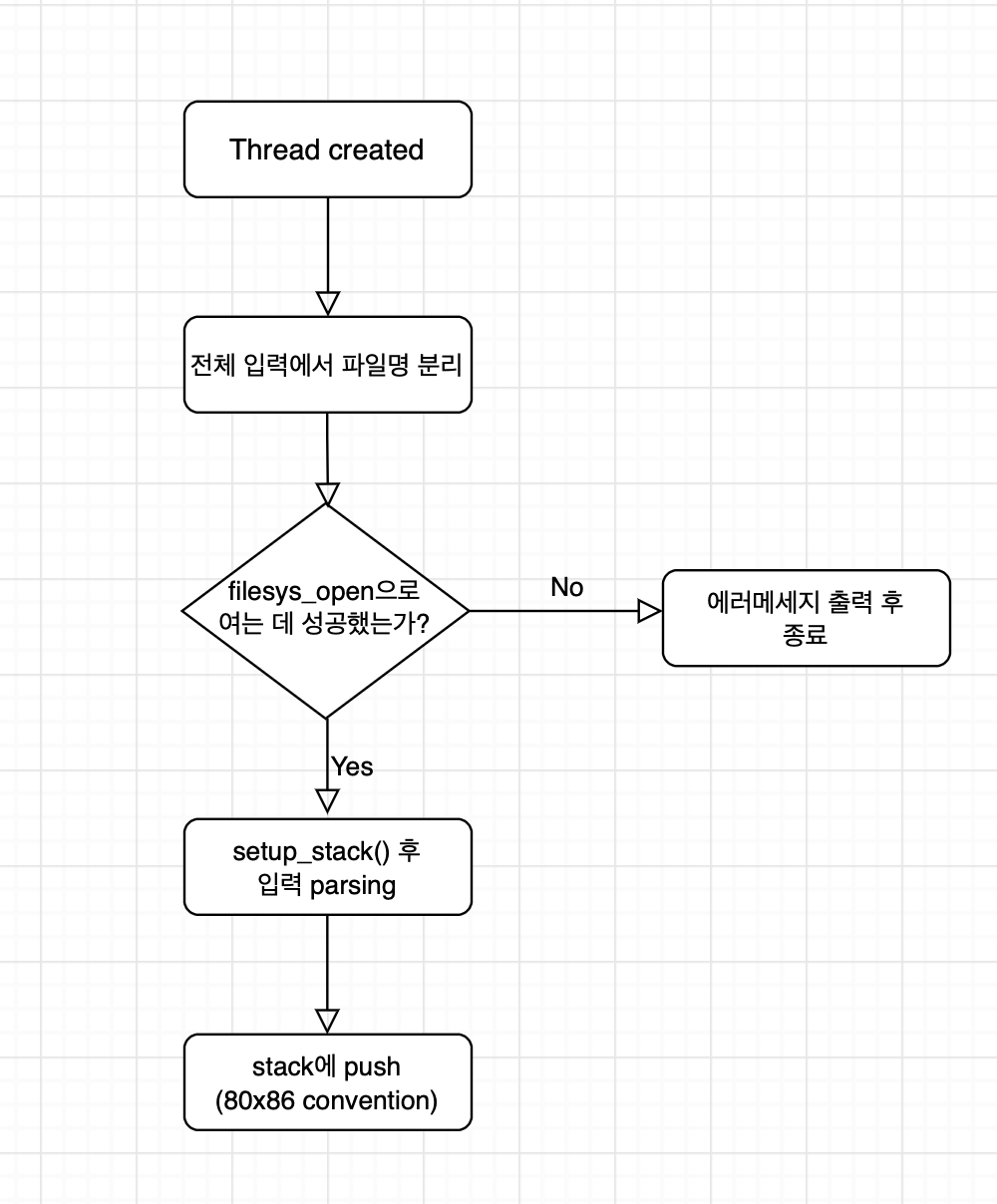
System call : userprog/syscall.c 에서 syscall\_handler 함수를 수정한다. switch-case를 사용하여 syscall number에 따라 알맞은 기능이 수행되도록 구현한다.

Thread : threads/thread.h 에 정의된 struct thread에 process wait에 필요한 변수(exit\_status, semaphore 등)을 추가한다.

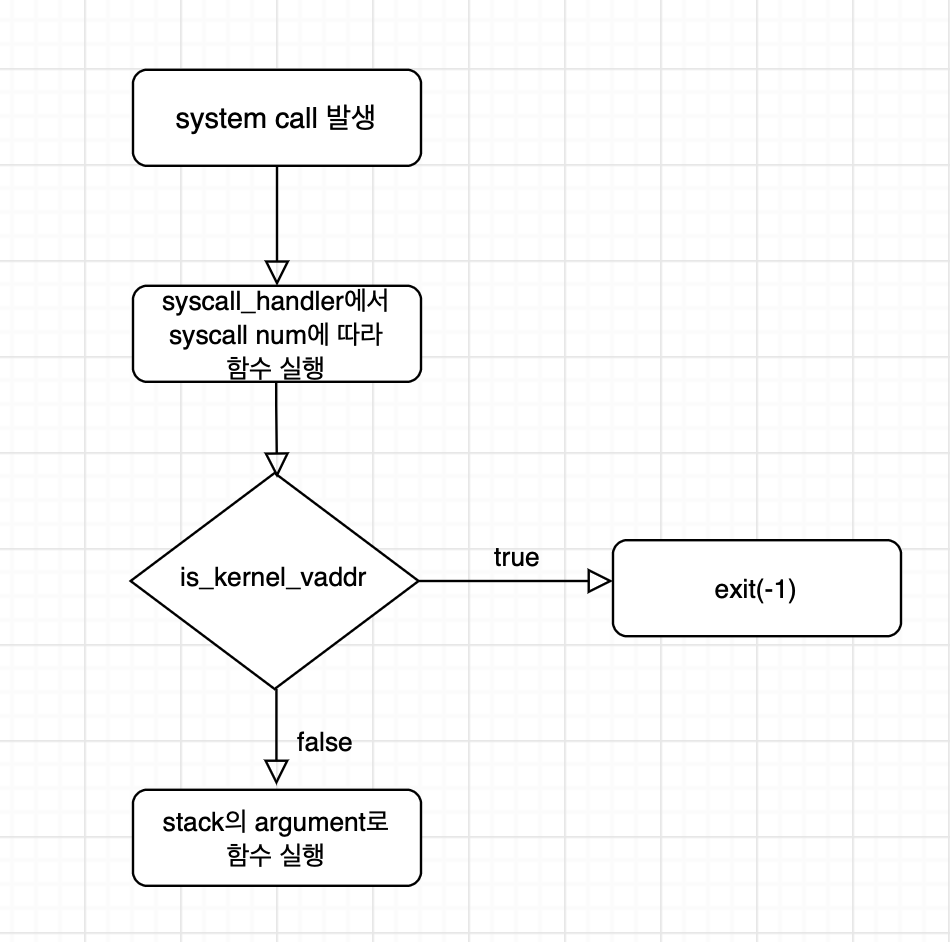
1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

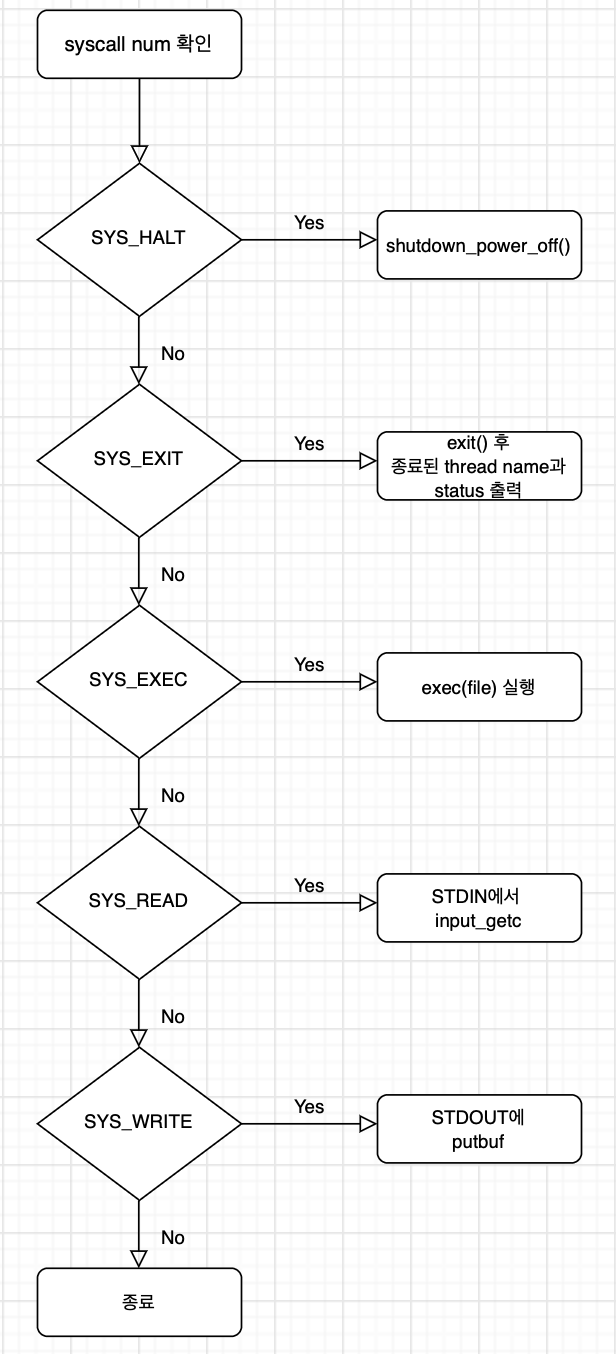
1. Argument Passing



1. User Memory Access



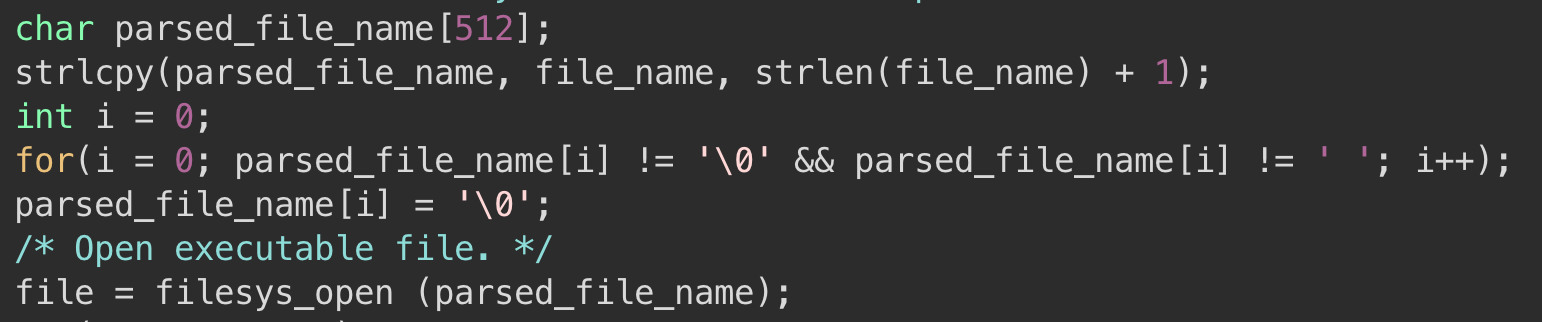
1. System Calls



* 1. **제작 내용**
* **II. B. 개발 내용의 실질적인 구현에 대해 코드 관점에서 작성.**
* **구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명.**
* **개발상 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결책에 대해 설명.**

1. Argument Passing

src/userprog/process.c의 load 함수에서 진행된다.



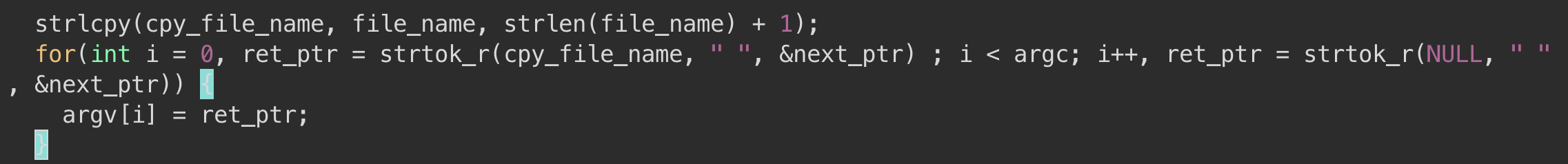
입력받은 argument 전체에서 맨 앞의 실제 file name을 parsed\_file\_name에 담는 부분이다. file\_name을 선회하며 처음 만나는 공백을 \0으로 바꾸어 file name만 구분하여 filesys\_open을 수행한다.

이후 setup\_stack이 완료된 후, stack에 내용을 담는다.

텍스트이(가) 표시된 사진

자동 생성된 설명

원본 string을 보존하기 위해서 cpy\_file\_name에 file\_name을 복사해서 사용하였고, strtok\_r을 진행한 횟수 만큼 argc에 argument의 개수를 담았다. argv는 argument를 담는 배열로 이중 포인터를 사용하여 2차원 배열처럼 사용하였다.



이후엔 file\_name 전체를 tokenize 하며 argument의 내용을 argv[]에 저장한다.

이후엔 본격적으로 stack에 내용물을 담는다.

텍스트이(가) 표시된 사진

자동 생성된 설명

argc의 개수 만큼 반복문을 돌며 stack에 argv[]의 내용을 담는다. 이 때 stack에 담기는 argument의 순서를 고려하여 index가 큰 순서로 담아준다. 담아준 후엔 argument가 있던 곳에 주소를 저장해놓는다(추후에 stack에 push하기 위함). argument들의 길이를 합산하는 total\_len 변수를 이용하여 word-align을 위한 길이만큼 esp를 이동해준다.

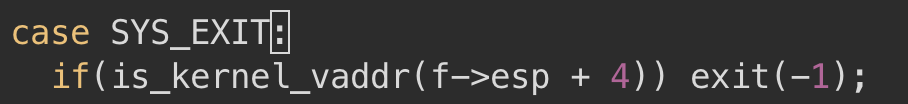
텍스트이(가) 표시된 사진

자동 생성된 설명

이후엔 차례대로 NULL sentinel과 argv[]의 주소, argv의 주소와 argc의 값, return address를 push한다.

1. User Memory Access

System call에 해당하는 함수를 실행하기 위하여 stack의 parameter를 넘겨줄 때 해당 stack의 pointer가 kernel 영역을 침범하고 있는 지 확인하는 부분을 각 system call 마다 추가하였다.



위와 같이 system call로 stack address에 접근할 때 해당 virtual address가 user 영역이 아니라면 exit(-1)을 수행하여 종료하도록 하였다.

1. System Calls

* **이번 프로젝트에서 개발한 시스템 콜을 구현 관점에서 상세히 서술.**

halt()

텍스트이(가) 표시된 사진

자동 생성된 설명

return 값과 parameter 없이 system call이 도착하면 shutdown\_power\_off 함수를 실행하도록 하였다.

exit(int status)

텍스트이(가) 표시된 사진

자동 생성된 설명

현재 thread의 이름과 상태를 출력하고, thread의 exit\_status를 업데이트 한 후 thread\_exit()으로 종료하도록 하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이를 위해 struct thread에 필요한 요소들을 추가하였다. 종료 상태를 포함하는 exit\_status와 wait 시에 필요한 두 semaphore, child list를 위한 list와 list\_elem을 추가하였다. 부모 thread를 가리키는 포인터와 두 flag용 bool은 실제로 사용하지는 않았으나 추후에 필요할 수 있으니 남겨두었다.

pid\_t exec(const char\* file)

텍스트이(가) 표시된 사진

자동 생성된 설명

전달받은 file을 이용하여 process\_execute 함수를 실행하고 생성된 thread의 pid(=tid)를 return 한다. 주석 처리 되어있는 부분은 실제로 메모리에 thread가 load 되었는지를 체크하는 부분이었으나 해당 코드가 없어도 정상적으로 동작하여 현재는 주석으로 남겨두었다.

int wait(pid\_t pid)

텍스트이(가) 표시된 사진

자동 생성된 설명

전달받은 pid를 process\_wait의 인자로 넘겨주고 status를 return 한다. 우선 process.c에서 하나의 함수를 선언하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

get\_child\_process 함수는 인자로 받은 pid에 해당하는 child thread의 포인터를 return 하고, 찾지 못했으면 NULL을 return 한다. 이는 child list를 순회하며 하나씩 비교해보는 방식으로 진행하였다. struct list를 통해 상위 구조체의 내용을 볼 수 있는 list\_entry 함수를 사용하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

원하는 child process를 wait하기 위해 구현한 함수를 사용하여 thread 포인터를 찾아내고, 만약 해당 포인터가 NULL이라면 -1을 return 한다. 제대로 찾았다면 child process의 수행을 기다리기 위해서 sema\_down을 실행한다. child process는 수행 후 process\_exit의 마지막 단계에서 sema\_up을 해준다. 그 직후 바로 다른 semaphore(sem\_load)를 down 해주는데 이는 child process를 list에서 제거하기 전에 child process가 먼저 종료되는 현상을 막기 위함이다. 정상적으로 child process를 종료하고 list에서 제거하였다면 exit\_status를 return 한다.

read와 write

텍스트이(가) 표시된 사진

자동 생성된 설명

Project 1에서는 STDIN/STDOUT에서의 입출력만 고려하기 때문에 해당 file descriptor에서의 입출력을 각각 input\_getc와 putbuf 함수로 구현하였다.

1. Additional System calls

* **새로운 시스템 콜(fibonacci, max\_of\_four\_int)을 구현하기 위해 수정하거나 작성한 코드에 대해 서술**

우선 두 system call을 위한 syscall number를 추가하기 위해서 lib/syscall-nr.h 파일을 수정하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

최하단에 새로운 SYS\_FIBO와 SYS\_MAX를 추가하였다. 또 max\_of\_four\_int의 경우 인자 4개를 필요로 하는 system call이기 때문에 lib/user/syscall.c에 syscall4를 정의하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

형식은 다른 syscallN을 참고하였다. 그 후 같은 파일에서 system call을 호출하는 부분을 추가하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이후엔 userprog/syscall.c에서 실제로 처리해주는 부분을 구현하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

마지막으로 syscall\_handler에서 두 system call에 대한 처리 부분을 추가하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* 1. **시험 및 평가 내용**
* **fibonacci 및 max\_of\_four\_int 시스템 콜 수행 결과를 캡처하여 첨부.**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

새롭게 추가한 system call이 잘 동작하는 모습이다.